System V Application Binary Interface

x86-64™ Architecture Processor Supplement

Draft Version 0.10

Edited by
Jan Hubicka[1], Andreas Jaeger[2], Mark Mitchell[3]

January 14, 2001

[1]jh@suse.cz
[2]aj@suse.de
[3]mark@codesourcery.com

Contents

1 Introduction **5**
1.1 Differences from the Intel386 ABI 5

2 Software Installation **7**

3 Low Level System Information **8**
3.1 Machine Interface 8
3.1.1 Processor Architecture 8
3.1.2 Data Representation 8
3.2 Function Calling Sequence 11
3.2.1 Registers and the Stack Frame 11
3.2.2 The Stack Frame 11
3.2.3 Parameter Passing 12
3.2.4 Functions Returning Scalars or No Value 14
3.2.5 Functions Returning Structures or Unions 14
3.3 Operating System Interface 16
3.3.1 Virtual Address Space 16
3.3.2 Page Size 16
3.3.3 Virtual Address Assignments 16
3.4 Coding Examples 18
3.4.1 Architectural Constraints 18
3.4.2 Position-Independend Function Prologue 20
3.4.3 Data Objects 20
3.4.4 Function Calls 20
3.4.5 Variable Argument Lists 21

4 Object Files **25**
4.1 ELF Header 25

4.1.1 Machine Information 25
4.2 Sections 25
4.3 Symbol Table 26
4.4 Relocation 26
4.4.1 Relocation Types 26

5 Program Loading and Dynamic Linking 29
5.1 Program Loading 29
5.2 Dynamic Linking 29
5.2.1 Program Interpreter 32
5.2.2 Initialization and Termination Functions 32

6 Libraries 33
6.1 C Library 33
6.1.1 Global Data Symbols 33

7 Development Environment 34

8 Execution Environment 35

9 Conventions 36
9.1 GOT pointer and IP relative addressing 36
9.2 Execution of 32bit programs 36
9.3 C++ 37

List of Tables

4.1 x86-64 Identification 25

List of Figures

3.1 Scalar Types 9
3.2 Bit-Field Ranges 10
3.3 Stack Frame 12
3.4 Generic Register Usage 15
3.5 Virtual Address Configuration 17
3.6 Conventional Segment Arrangements 17
3.7 Position-Independent Direct Function Call 20
3.8 Position-Independent Indirect Function Call 21
3.9 register save area 22
3.10 `va_list` type declaration 22
3.11 Sample implementation of `va_arg(l,int)` 24

4.1 Relocatable Fields 26
4.2 Relocation Types 27

5.1 Global Offset Table 29
5.2 Procedure Linkage Table 30

Chapter 1

Introduction

The x86-64 architecture [1] is an extension of the x86 architecture. Any processor implementing the x86-64 architecture specification will also provide compatiblity modes for previous descendants of the Intel 8086 architecture, including 32-bit processors such as the Intel 386, Intel Pentium, and AMD K6-2 processor. Operating systems conforming to the x86-64 ABI may provide support for executing programs that are designed to execute in these compatiblity modes. The x86-64 ABI does not apply to such prorams; this document applies only programs running in the "long" mode provided by the x86-64 architecture.

Except where otherwise noted, the x86-64 architecture ABI follows the conventions described in the Intel386 ABI. Rather than replicate the entire contents of the Intel386 ABI, the x86-64 ABI indicates only those places where changes have been made to the Intel386 ABI.

No attempt has been made to specify an ABI for languages other than C. However, it is assumed that many programming languages will wish to link with code written in C, so that the ABI specifications documented here are relevant.[2]

1.1 Differences from the Intel386 ABI

The most fundamental differences from the Intel386 ABI document are as follows:

- Sizes of fundamental data types.

[1] The architecture specification is available on the web at `http://www.x86-64.org/documentation`.

[2] See section 9.3 for details on C++ ABI.

- Parameter-passing conventions.
- Floating-point computations.
- Removal of the GOT register.
- Use of RELA relocations.

Chapter 2

Software Installation

No changes required.

Chapter 3

Low Level System Information

3.1 Machine Interface

3.1.1 Processor Architecture

3.1.2 Data Representation

Within this specification, the term *halfword* refers to a 16-bit object, the term *word* refers to a 32-bit object, the term *doubleword* refers to a 64-bit object, and the term *quadword* refers to a 128-bit object.

Fundamental Types

Figure 3.1 shows the correspondence between ISO C's scalar types and the processor's.

The `__float128` type uses a 15-bit exponent, a 113-bit mantissa (the high order significant bit is implicit) and an exponent bias of 16383.[1]

The `long double` type uses a 15 bit exponent, a 64-bit mantissa with an explicit high order significand bit and an exponent bias of 16383.[2] Although a `long double` requires 16 bytes of storage, only the first 10 bytes are significant. The remaining six bytes are tail padding, and the contents of these bytes are undefined.

[1] Initial implementations of the x86-64 architecture are expected to support operations on the `__float128` type only via software emulation.

[2] This type is the x87 double extented precision data type.

Figure 3.1: Scalar Types

Type	C	sizeof	Alignment (bytes)	x86-64 Architecture
Integral	`char` `signed char`	1	1	signed byte
	`unsigned char`	1	1	unsigned byte
	`short` `signed short`	2	2	signed halfword
	`unsigned short`	2	2	unsigned halfword

	`int`
	`signed int`
	`enum`
		4	4	signed word

	`unsigned int`	4	4	unsigned word

	`long`
	`signed long`
	`long long`
	`signed long long`
		8	8	signed doubleword

	`unsigned long` `unsigned long long`	8 8	8 8	unsigned doubleword unsigned doubleword
	`__int128` `signed __int128`	16 16	8 8	signed quadword signed quadword
	`unsigned __int128`	16	8	unsigned quadword
Pointer	*`any-type`* `*` *`any-type`* `(*)()`	8	8	unsigned doubleword
Floating-point	`float`	4	4	single (IEEE)
	`double`	8	8	double (IEEE)
	`long double`	16	16	80-bit extended (IEEE)
	`__float128`	16	16	128-bit extended (IEEE)
Packed	`__m64`	8	8	*MMX* and 3DNow!
	`__m128`	16	16	SSE and SSE-2

The `__int128` type is stored in little-endian order in memory, i.e., the 64 low-order bits are stored at a a lower address than the 64 high-order bits.

A null pointer (for all types) has value zero.

Like the Intel386 architecture, the x86-64 architecture does not require all data access to be properly aligned. Accessing misaligned data will be slower than accessing properly aligned data, but otherwise there is no difference.

Aggregates and Unions

An array uses the same alignment as its elements, except that a local or global array variable that requires at least 16 bytes, or a C99 local or global variable-length array variable, always has alignment of at least 16 bytes.[3]

No other changes required.

Bit-Fields

Amend the description of bit-field ranges as follows:

Figure 3.2: Bit-Field Ranges

Bit-field Type	Width w	Range
`signed long`	1 to 64	-2^{w-1} to $2^{w-1}-1$
`long`		0 to 2^w-1
`unsigned long`		0 to 2^w-1

The ABI does not permit bitfields having the type `__m64` or `__m128`. Programs using bitfields of these types are not portable.

No other changes required.

[3]The alignment requirement allows the use of SSE instructions when operating on the array. The compiler cannot in general calculate the size of a variable-length array, but it is expected that most VLAs will require at least 16 bytes, so it is logical to mandate that VLAs have at least a 16-byte alignment.

3.2 Function Calling Sequence

This section describes the standard function calling sequence, including stack frame layout, register usage, parameter passing and so on.

The standard calling sequence requirements apply only to global functions. Local functions that are not reachable from other compilation units may use different conventions. Nevertheless, it is recommended that all functions use the standard calling sequence when possible.

3.2.1 Registers and the Stack Frame

The x86-64 architecture provides 16 general purpose 64-bit registers. In addition the architecture provides 16 SSE registers, each 128 bits wide and 8 x87 floating point registers, each 80 bits wide. Each of the x87 floating point registers may be referred to in *MMX*/3DNow! mode as a 64-bit register. All of these registesr are global to all procedures in a running program.

This subsection discusses usage of each register. Registers `%rbp`, `%rbx` and `%r12` through `%r15` "belong" to the calling function and the called function is required to preserve their values. In other words, a called function must preserve these registers' values for its caller. Remaining registers "belong" to the called function.[4] If a calling function wants to preserve such a register value across a function call, it must save the value in its local stack frame.

The CPU shall be in MMX mode upon entry to a function. Therefore, every function that uses the x87 register stack is required to issue an `emms` or `femms` instruction before accessing the x87 register stack.[5] The direction flag in the `%eflags` register must be clear on function entry, and on function return.

3.2.2 The Stack Frame

In addition to registers, each function has a frame on the run-time stack. This stack grows downwards from high addresses. Figure 3.3 shows the stack organization.

The end of the input argument area shall be aligned on a 16 byte boundary. In other words, the value $(\texttt{\%rsp} - 8)$ is always a multiple of 16 when control is

[4] Note that in contrast to the Intel386 ABI, `%rdi`, and `%rsi` belong to the called function, not the caller.

[5] All *MMX* registers are caller-saved, so callees that make use of the x87 register stack may use the faster `femms` instruction.

Figure 3.3: Stack Frame

Position	Contents	Frame
8n+16(%rbp)	argument doubleword n	Previous
	...	
16(%rbp)	argument doubleword 0	
8(%rbp)	return address	Current
0(%rbp)	previous %rbp value	
-8(%rbp)	unspecified	
	...	
0(%rsp)	variable size	
128(%rsp)	red zone	

transferred to the function entry point. The stack pointer, %rsp, always points to the end of the latest allocated stack frame. [6]

The 128-byte area beyond the location pointed to by %rsp is considered to be reserved and shall not be modified by signal or interrupt handlers.[7] Therefore, functions may use this area for temporary data that is not needed across function calls. In particular, leaf functions may use this area for their entire stack frame, rather than adjusting the stack pointer in the prologue and epilogue.

3.2.3 Parameter Passing

After the argument values have been computed, they are placed in registers, or pushed on the stack. The arguments are processed in right-to-left order. Since the stack grows downwards, the rightmost argument will have the highest address.[8]

For each argument, the following method is used to determine the location in which the argument is passed. In all cases where a value is pushed on the stack,

[6] The conventional use of %rbp as a frame pointer for the stack frame may be avoided by using %rsp (the stack pointer) to index into the stack frame. This technique saves two instructions in the prologue and epilogue and makes one additional general-purpose register (%rbp) available.

[7] Locations within 128 bytes can be addressed using one-byte displacements.

[8] Right-to-left order makes the handling of functions that take a variable number of arguments simpler. The location of the first argument can always be computed statically, based on the type of that argument. It would be difficult to compute the address of the first argument if the arguments were pushed in left-to-right order.

the stack shall be left doubleword-aligned. In all cases, the value pushed shall be located on a boundary whose alignment is the maximum of 8 (doubleword alignment) and the alignment of the type being pushed. If the value does not consume the entire doubleword, the contents of the unused storage located at higher numbered addresses within the doubleword is undefined.

If the argument is a scalar type, other than `__m64` or `__int128`, then it is placed in the next available register suitable for its type. In particular, arguments of integral or pointer type are placed in the next available general purpose register, taken in the order `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, and `%r9`. [9] Arguments of floating point type and of type `__m128`, are placed in the next available SSE register, taken in order from `%xmm0` to `%xmm15`. If no registers are available, the values are placed on the stack.

Scalar values of type `__int128` are handled almost as if they consisted of two separate 64-bit arguments of type `long`. The 64 low-order bits are considered to be the first argument, and therefore processed second.[10] If there are not enough registers available to allow passing both arguments in registers, then the value is passed on the stack.

Scalar values of type `__m64` are always placed on the stack.

Structure or union objects with more than 16 bytes, those that contain scalar components of type `__m64`, or, in C++, non-POD[11] structure or union types,[12] are always passed on the stack. The stack is aligned as required by the alignment of the structure or union type being passed. Then, the structure value is copied onto the stack.

Structure or union objects that contain no more than 16 bytes and, in C++, are PODs, are passed in registers, if registers are available. If the entire structure

[9] Note that `%r11` is neither required to be preserved, nor is it used to pass arguments. Making this register available as scratch register means that code in the PLT need not spill any registers when computing the address to which control needs to be transferred. `%rax` is used to indicate the number of SSE arguments passed to a function requiring a variable number of arguments. `%r10` is used for passing a function's static chain pointer.

[10] Thus, if the value is placed on the stack, the low-order bits will be at a lower address, which allows the caller to access the value normally.

[11] The term POD is from the ANSI/ISO C++ Standard, and stands for Plain Ol' Data. Although the exact definition is technical, a POD is essentially a structure or union that could could have been written in C; there cannot be any member functions, or base classes, or similar C++ extensions.

[12] A non-POD object cannot be returned in registers because such objects must have well defined addresses; the address at which an object is constructed (by the caller) and the address at which the object is destroyed (by the callee) must be the same. Similar issues apply when returning a non-POD object from a function.

cannot be placed in registers as described below, then the structure is passed on the stack, as above.

If the structure contains a single scalar component, then it is passed as if it were a single argument of that scalar type.

Otherwise, the first (or only, if the structure has no more than 8 bytes), doubleword of the structure is passed first. If there is only one component, the argument is passed as if it were an argument of that type. Otherwise, if all scalar components in the first component are of floating point type, the first doubleword is passed as if it were an argument of floating point type. If all the components begin at the same address, then the component is passed as if it were an argument of the longest floating point type used by these components. Otherwise, the argument is passed as if it were an argument of type `__float128`.[13]

Otherwise, the doubleword is passed as if it were a single integer argument, using the smallest type sufficient to contain the argument. Any bytes not used by the value passed have undefined contents.

If the structure has more than 8 bytes, then the process is repeated for the second doubleword.

3.2.4 Functions Returning Scalars or No Value

If a function returns a scalar of type `__m64`, the value is returned in `%mm0`. If the function returns a scalar of some other type, the value is returned in a register or registers. The register or registers chosen are the same as those that would be selected for a value of the same type to be passed to a function taking only one argument of that same type, except that the list for integral or pointer type scalars consists of `%rax` and `%rdx`.

3.2.5 Functions Returning Structures or Unions

If a function returns a structure or union whose size is greater than 16 bytes, or, for C++, if the structure or union is a non-POD, then the caller provides space for the return value and passes the address of this storage in `%rdi` as if it were the first argument to the function. In effect, this address becomes a “hidden” first argument.

[13] In this case, there will be no padding bits, because this case can only occur if there is a value of type `float` in the second word of the doubleword.

Figure 3.4: Generic Register Usage

Register	Usage	Preserved across function calls
%rax	temporary register; with variable arguments passes information about the number of SSE registers used; 1st return register	No
%rbx	callee-saved register; optionally used as base pointer	Yes
%rcx	used to pass 4th integer argument to functions	No
%rdx	used to pass 3rd argument to functions ; 2nd return register	No
%rsp	stack pointer	Yes.
%rbp	callee-saved register	Yes
%rsi	used to pass 2nd argument to functions	No
%rdi	used to pass 1st argument to functions	No
%r8	used to pass 5th argument to functions	No
%r9	used to pass 6th argument to functions	No
%r10	temporary register, used for passing a function's static chain pointer	No
%r11	temporary register	No
%r12	callee-saved register	Yes
%r13	callee-saved register	Yes
%r14	callee-saved register	Yes
%r15	callee-saved register	Yes
%xmm0–%xmm1	used to pass and return floating point arguments	No
%xmm2–%xmm15	used to pass floating point arguments	No
%mmx0–%mmx1, %st0–%st1	temporary registers; used to return MMX arguments	No
%mmx2–%mmx8, %st2–%st8	temporary registers	No

Note: Register assignment is right to left.

In this case, the function also sets `%rdi` to the value of the original address in the callee's area before it returns. Thus when the caller receives control again, the address of the returned object resides in register `%rdi` and can be used to access the object.

If a function returns a structure or union whose size is less than or equal to 16 bytes, and, for C++, the structure or union is a POD, then the structure is returned in registers. The registers chosen for a given structure or union type are the same as those that would be selected for a value of the same type to be passed to a function taking only one argument of that same structure type, except that the list for integral or pointer type scalars consists of `%rax` and `%rdx`. For returning, structures and unions are treated the same way as for passing them.

3.3 Operating System Interface

3.3.1 Virtual Address Space

Although the x86-64 architecture uses 64-bit pointers, implementations are only required to handle 48-bit addresses. Therefore, conforming processes may only use addresses from `0x0000000000000000` to `0x00007fffffffffff`[14].

No other changes required.

3.3.2 Page Size

Systems are permitted to use any page size between 4KB and 64KB, inclusive.

No other changes required.

3.3.3 Virtual Address Assignments

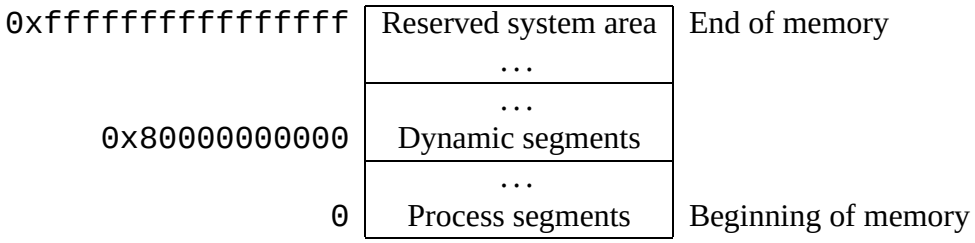
Conceptually processes have the full address space available. In practice, however, several factors limit the size of a process.

- The system reserves a configuration dependent amount of virtual space.
- The system reserves a configuration dependent amount of space per process.

[14] 0x0000ffffffffffff is not a canoncial address and cannot be used.

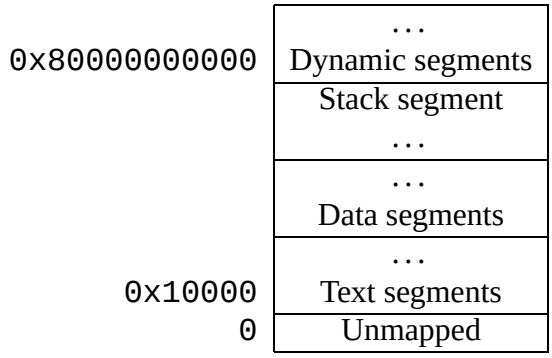
- A process whose size exceeds the system's available combined physical memory and secondary storage cannot run. Although some physical memory must be present to run any process, the system can execute processes that are bigger than physical memory, paging them to and from secondary storage. Nonetheless, both physical memory and secondary storage are shared resources. System load, which can vary from one program execution to the next, affects the available amount.

Figure 3.5: Virtual Address Configuration



Although applications may control their memory assignments, the typical arrangement appears in figure 3.6.

Figure 3.6: Conventional Segment Arrangements



3.4 Coding Examples

The following sections show only the difference to the i386 ABI.

3.4.1 Architectural Constraints

The x86-64 architecture usually does not allow to encode arbitrary 64bit constants as immediate operand of the instruction. Most instructions accept 32-bit immediates that are sign extended to the 64-bit ones. Additionally the 32-bit operations with register destinations implicitly perform zero extension making loads of 64-bit immediates with upper half set to 0 even cheaper.

Additionally the branch instructions accept 32-bit immediate operands that are sign extended and used to adjust instruction pointer. Similarly an instruction pointer relative addressing mode exists for data accesses with equivalent limitations.

In order to improve performance and reduce code size, it is desirable to use different code models depending on the requirements.

Code models define constraints for symbolic values that allow the compiler to generate better code. Basically code models differ in addressing (absolute versus position independent), code size, data size and address range. We define only a small number of code models that are of general interest:

Small code model The virtual address of code executed is known at link time. Additionally all symbols are known to be located in the virtual addresses in the range from 0 to $2^{31} - 2^{10} - 1$.

This allows the compiler to encode symbolic references with offsets in the range from -2^{31} to 2^{10} directly in the sign extended immediate operands, with offsets in the range from 0 to $2^{31} + 2^{10}$ in the zero extended immediate operands and use instruction pointer relative addressing for the symbols wit offsets in the range -2^{10} to 2^{10}.

This is the fastest code model and we expect it to be suitable for the vast majority of programs.

Kernel code model The kernel of an operating system is usually rather small but runs in the negative half of the address space. So we define all symbols to be in the range from $2^{64} - 2^{31}$ to $2^{64} - 2^{10}$.

This code model has advantages similar to those of the small model, but allows encoding of zero extended symbolic references only for offsets from

2^{31} to $2^{31} + 2^{10}$. The range offsets for sign extended reference changes to 0–$2^{31} + 2^{10}$.

Medium code model The medium code model does not make any assumptions about the range of symbolic references to data sections. Size and address of the text section have the same limits as the small code model.

This model requires the compiler to use `movabs` instructions to access static data and to load addresses into register, but keeps the advantages of the small code model for manipulation of addresses to the text section (specially needed for branches).

Large code model The large code model makes no assumptions about addresses and sizes of sections.

The compiler is required to use the `movabs` instruction, as in the medium code model, even for dealing with addresses inside the text section. Additionally, indirect branches are needed when branching to addresses whose offset from the current instruction pointer is unknown.

It is possible to avoid the limitation for the text section by breaking up the program into multiple shared libraries, so we do not expect this model to be needed in the foreseeable future.

Small position independent code model (PIC) Unlike the previous models, the virtual addresses of instructions and data are not known until dynamic link time. So all addresses have to be relative to the instruction pointer.

Additionally the maximum distance between a symbol and the end of an instruction is limited to $2^{31} - 2^{10} - 1$, allowing the compiler to use instruction pointer relative branches and addressing modes supported by the hardware for every symbol with an offset in the range -2^{10} to 2^{10}.

Medium position independent code model (PIC) This model is like the previous model, but makes no assumptions about the distance of symbols to the data section.

In the medium PIC model, the instruction pointer relative addressing can not be used directly for accessing static data, since the offset can exceed the limitations on the size of the displacement field in the instruction. Instead an unwind sequence consisting of `movabs`, `lea` and `add` needs to be used.

Large position independent code model (PIC) This model is like the previous model, but makes no assumptions about the distance of symbols.

The large PIC model implies the same limitation as the medium PIC model regarding addressing of static data. Additionally, references to the global offset table and to the procedure linkage table and branch destiniations need to be calculated in a similar way.

3.4.2 Position-Independend Function Prologue

x86-64 does not need any function prologue for calculating the global offset table address since it does not have an explicit GOT pointer.

3.4.3 Data Objects

Not done yet.

3.4.4 Function Calls

Figure 3.7: Position-Independent Direct Function Call

extern void function (); function ();	.globl function call function@PLT

Figure 3.8: Position-Independent Indirect Function Call

extern void (*ptr) ();
extern void name ();
ptr = name;
 	
(*ptr)();
	.globl ptr, name
	
	movl ptr@GOTPCREL(%rip), %rax
	movl name@GOTPCREL(%rip), %rdx
	movl %rdx, (%rax)
	
	movl ptr@GOTPCREL(%rip), %rax
	call *(%rax)

3.4.5 Variable Argument Lists

Some otherwise portable C programs depend on the argument passing scheme, implicitly assuming that 1) all arguments are passed on the stack, and 2) arguments appear in increasing order on the stack. Programs that make these assumptions never have been portable, but they have worked on many implementations. However, they do not work on the x86-64 architecture because some arguments are passed in registers. Portable C programs must use the header files `<stdarg.h>` or `<varargs.h>` in order to handle variable argument lists.

When a function taking variable-arguments is called, `%rax` must be set to eight times the number of floating point parameters passed to the function in SSE registers.

The Register Save Area

The prologue of a function taking a variable argument list and known to call the macro `va_start` is expected to save the argument registers to the *register save area*. Each argument register has a fixed offset in the register save area as defined in the figure 3.9.

Only registers possibly used by argument passed as variable arguments needs to be saved. Other fields may not be accessed and can be used for other purposes. In the case a function is known to never accept arguments passed in registers[15], the register save area may be omitted entirely.

[15]This fact may be determined either by exploring types used by the `va_arg` macro, or by the fact that the named arguments already are exhausted the argument registers entirely

The prologue should use `%rax` to avoid saving of unnecesary XMM registers. This is especially important to avoid integer only programs from initializing the XMM unit.

Figure 3.9: Register Save Area

register	offset
`%rdi`	0
`%rsi`	8
`%rdx`	16
`%rcx`	24
`%r8`	32
`%r9`	40
`%xmm0`	48
`%xmm1`	64
...	
`%xmm15`	288

The `va_list` Type

The `va_list` type is an array containing a single element of one structure containing the necessary information to implement the `va_arg` macro. The C definition of `va_list` type is given in figure 3.10.

Figure 3.10: `va_list` Type Declaration

```
typedef struct {
   unsigned int gp_offset;
   unsigned int fp_offset;
   void *overflow_arg_area;
   void *reg_save_area;
} va_list[1];
```

The `va_start` Macro

The `va_start` macro initializes the structure as follows:

reg_save_area The element points to the start of the register save area.

overflow_arg_area This pointer is used to fetch arguments passed on the stack. It is initialized to the address of the first argument passed on the stack, if any, and then always updated to point to the start of the next stack argument.

gp_offset The element holds the offset in bytes from `reg_save_area` to the place where the next available general purpose argument register is saved. In case all argument registers have been exhausted, it is set to the value 48 ($6 * 8$).

fp_offset The element holds the offset in bytes from `reg_save_area` to the place where the next available floating point argument register is saved. In case all argument registers have been exhausted, it is set to the value 304 ($6 * 8 + 16 * 16$).

The `va_arg` Macro

The algorithm for the generic `va_arg(l, type)` implementation is defined as follows:

1. Determine whether `type` may be passed in the registers. If not go to step 7.

2. Compute `num_gp` to hold the number of general purpose registers needed to pass `type` and `num_fp` to hold the number of floating point registers needed.

3. Verify whether arguments fits into registers. In the case:

$$\texttt{l->gp\_offset} > 48 - \texttt{num\_gp} * 8$$

   or

$$\texttt{l->fp\_offset} > 304 - \texttt{num\_fp} * 16$$

   go to step 7.

4. Fetch `type` from `l->reg_save_area` offsetted by `l->gp_offset` and/or `l->fp_offset`. This may require copying to a temporary location in case the parameter is passed in different register classes or requires an alignment greater than 8 for general purpose registers and 16 for XMM registers.

5. Set:
$$\texttt{l->gp\_offset} = \texttt{l->gp\_offset} + \texttt{num\_gp} * 8$$
$$\texttt{l->fp\_offset} = \texttt{l->fp\_offset} + \texttt{num\_fp} * 16.$$

6. Return the fetched `type`.

7. Align `l->overflow_arg_area` upwards to a 16 byte boundary if alignment needed by `type` exceeds 8 byte boundary.

8. Fetch `type` from `l->overflow_arg_area`.

9. Set `l->overflow_arg_area` to:
$$\texttt{l->overflow\_arg\_area} + \texttt{sizeof}(\texttt{type})$$

10. Align `l->overflow_arg_area` upwards to an 8 byte boundary.

11. Return the fetched `type`.

The `va_arg` macro is usually implemented as a compiler builtin and expanded in simplified forms for each particular type. Figure `%fig-va_arg` is a sample implementation of the `va_arg` macro.

Figure 3.11: Sample Implementation of `va_arg(l, int)`

```
        movl    l->gp_offset, %eax
        cmpl    $48, %eax               Is register available?
        jae     stack                   If not, use stack
        leal    $8(%rax), %edx          Next available register
        addq    l->reg_save_area, %rax  Address of saved register
        movl    %edx, l->gp_offset      Update gp_offset
        jmp     fetch
stack:  movq    l->overflow_arg_area, %rax  Address of stack slot
        leaq    8(%rax), %rdx           Next available stack slot
        movq    %rdx,l->overflow_arg_area   Update
fetch:  movl    (%rax), %eax            Load argument
```

Chapter 4

Object Files

4.1 ELF Header

4.1.1 Machine Information

For file identification in `e_ident`, the x86-64 architecture requires the following values.

Table 4.1: x86-64 Identification

Position	Value
`e_ident[EI_CLASS]`	`ELFCLASS64`
`e_ident[EI_DATA]`	`ELFDATA2LSB`

Processor identification resides in the ELF header's `e_machine` member and must have the value `EM_X86_64`.[1]

4.2 Sections

No changes required.

[1] The value of this identifier is 62.

4.3 Symbol Table

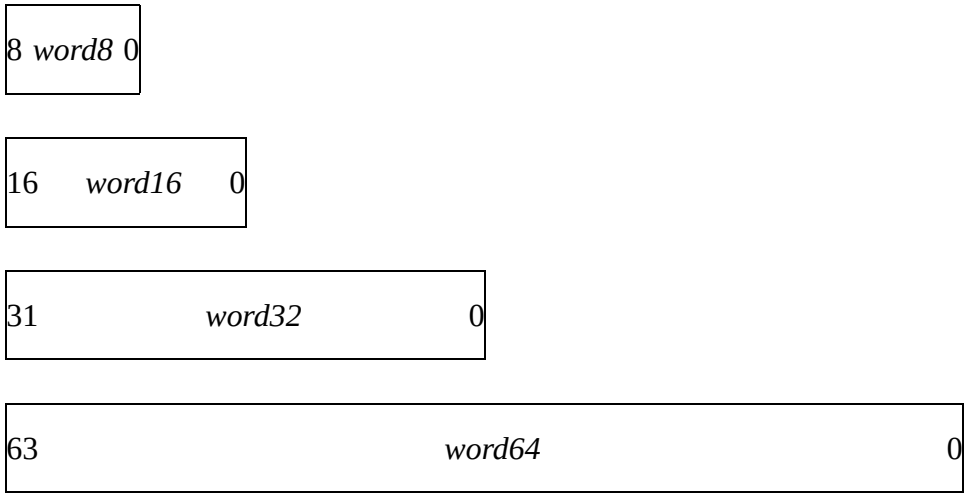
No changes required.

4.4 Relocation

4.4.1 Relocation Types

The x86-64 ABI adds one additional field:

Figure 4.1: Relocatable Fields



word8	This specifies a 8-bit field occupying 1 byte.
word16	This specifies a 16-bit field occupying 2 bytes with arbitrary byte alignment. These values use the same byte order as other word values in the x86-64 architecture.
word32	This specifies a 32-bit field occupying 4 bytes with arbitrary byte alignment. These values use the same byte order as other word values in the x86-64 architecture.
word64	This specifies a 64-bit field occupying 8 bytes with arbitrary byte alignment. These values use the same byte order as other word values in the x86-64 architecture.

The x86-64 ABI architectures uses only `Elf64_Rela` relocation entries with explicit addends. The `r_addend` member serves as the relocation addend.

Figure 4.2: Relocation Types

Name	Value	Field	Calculation
R_X86_64_NONE	0	none	none
R_X86_64_64	1	*word64*	S + A
R_X86_64_PC32	2	*word32*	S + A - P
R_X86_64_GOT32	3	*word32*	G + A
R_X86_64_PLT32	4	*word32*	L + A - P
R_X86_64_COPY	5	none	none
R_X86_64_GLOB_DAT	6	*word64*	S
R_X86_64_JUMP_SLOT	7	*word64*	S
R_X86_64_RELATIVE	8	*word64*	B + A
R_X86_64_GOTPCREL	9	*word32*	G + GOT + A - P
R_X86_64_32	10	*word32*	S + A
R_X86_64_32S	11	*word32*	S + A
R_X86_64_16	12	*word16*	S + A
R_X86_64_PC16	13	*word16*	S + A - P
R_X86_64_8	14	*word8*	S + A
R_X86_64_PC8	15	*word8*	S + A - P

The special semantics for most of these relocation types are identical to those

used for the Intel386 architecture. [2] [3]

The R_X86_64_GOTPCREL relocation has different semantics from the i386 R_I386_GOTPC relocation. In particular, because the x86-64 architecture has an addressing mode relative to the instruction pointer, it is possible to load an address from the GOT using a single instruction. The calculation done by the R_X86_64_GOTPCREL relocation gives the difference between the location in the GOT where the symbol's address is given and the location where the relocation is applied.

The R_X86_64_32 and R_X86_64_32S relocations truncate the computed value to 32-bits. The linker must verify that the generated value for the R_X86_64_32 (R_X86_64_32S) relocation zero-extends (sign-extends) to the original 64-bit value.

A program or object file using R_X86_64_8, R_X86_64_16, R_X86_64_PC16 or R_X86_64_PC8 relocations is not conformant to this ABI, these relocations are only added for documentation purposes. The R_X86_64_16, and R_X86_64_8 relocations truncate the computed value to 16-bits resp. 8-bits.

[2] Even though the x86-64 architecture supports IP-relative addressing modes, a GOT is still required since the address from a particular instruction to a particular data item cannot be known by the static linker.

[3] Note that the x86-64 architecture assumes that offsets into GOT are 32-bit values, not 64-bit values. This choice means that a maximum of $2^{32}/8 = 2^{29}$ entries can be placed in the GOT. However, that should be more than enough for most programs. In the event that it is not enough, the linker could create multiple GOTs. Because 32-bit offsets are used, loads of global data do not require loading the offset into a displacement register; the base plus immediate displacement addressing form can be used.

Chapter 5

Program Loading and Dynamic Linking

5.1 Program Loading

No changes required.

5.2 Dynamic Linking

Dynamic Section

No changes required.

Global Offset Table

The global offset table contains 64-bit addresses.

No other changes required.

Figure 5.1: Global Offset Table

```
extern Elf64_Addr _GLOBAL_OFFSET_TABLE_ [];
```

Function Addresses

No changes required.

Procedure Linkage Table

Much as the global offset table redirects position-independent address calculations to absolute locations, the procedure linkage table redirects position-independent function calls to absolute locations. The link editor cannot resolve execution transfers (such as function calls) from one executable or shared object to another. Consequently, the link editor arranges to have the program transfer control to entries in the procedure linkage table. On the x86-64 architecture, procedure linkage tables reside in shared text, but they use addresses in the private global offset table. The dynamic linker determines the destinations' absolute addresses and modifies the global offset table's memory image accordingly. The dynamic linker thus can redirect the entries without compromising the position-independence and shareability of the program's text. Executable files and shared object files have separate procedure linkage tables. Unlike Intel386 ABI, this ABI uses the same procedure linkage table for both programs and shared objects.

Figure 5.2: Procedure Linkage Table

```
.PLT0:  pushq   GOT+8(%rip); GOT[1]
        jmp     GOT+16(%rip) ; GOT[2]
        nop
        nop
        nop
        nop
.PLT1:  jmp     *name1@GOTPC(%rip)
        pushq   $index
        jmp     .PLT0@PC
.PLT2:  jmp     *name2@GOTPC(%rip)
        pushq   $index
        jmp     .PLT0@PC
        ...
```

Following the steps below, the dynamic linker and the program "cooperate"

to resolve symbolic references through the procedure linkage table and the global offset table.

1. When first creating the memory image of the program, the dynamic linker sets the second and the third entries in the global offset table to special values. Steps below explain more about these values.

2. Each shared object file in the process image has its own procedure linkage table, and control transfers to a procedure linkage table entry only from within the same object file.

3. For illustration, assume the program calls `name1`, which transfers control to the label `.PLT1`.

4. The first instruction jumps to the address in the global offset table entry for `name1`. Initially the global offset table holds the address of the following `pushl` instruction, not the real address of `name1`.

5. Now the program pushes a relocation index (*index*) on the stack. The relocation index is a 32-bit, non-negative index into the relocation table addressed by the `DT_JMPREL` dynamic section entry. The designated relocation entry will have type `R_X86_64_JUMP_SLOT` , and its offset will specify the global offset table entry used in the previous `jmp` instruction. The relocation entry contains a symbol table index that will reference the appropriate symbol, `name1` in the example.

6. After pushing the relocation index, the program then jumps to `.PLT0`, the first entry in the procedure linkage table. The `pushl` instruction places the value of the second global offset table entry (GOT+8) on the stack, thus giving the dynamic linker one word of identifying information. The program then jumps to the address in the third global offset table entry (GOT+16), which transfers control to the dynamic linker.

7. When the dynamic linker receives control, it unwinds the stack, looks at the designated relocation entry, finds the symbol's value, stores the "real" address for `name1` in its global offset table entry, and transfers control to the desired destination.

8. Subsequent executions of the procedure linkage table entry will transfer directly to `name1`, without calling the dynamic linker a second time. That

is, the `jmp` instruction at `.PLT1` will transfer to `name1`, instead of "falling through" to the `pushl` instruction.

The `LD_BIND_NOW` environment variable can change the dynamic linking behavior. If its value is non-null, the dynamic linker evaluates procedure linkage table entries before transferring control to the program. That is, the dynamic linker processes relocation entries of type `R_X86_64_JUMP_SLOT` during process initialization. Otherwise, the dynamic linker evaluates procedure linkage table entries lazily, delaying symbol resolution and relocation until the first execution of a table entry.

5.2.1 Program Interpreter

There is one valid program interpreter for programs conforming to the x86-64 ABI:

```
/lib/ld64.so.1
```

5.2.2 Initialization and Termination Functions

The implementation is responsible for executing the initialization functions specified by `DT_INIT` , `DT_INIT_ARRAY` , and `DT_PREINIT_ARRAY` entries in the executable file and shared object files for a process, and the termination (or finalization) functions specified by `DT_FINI` and `DT_FINI_ARRAY` , as specified by the *System V ABI*. The user program plays no further part in executing the initialization and termination functions specified by these dynamic tags.

Chapter 6

Libraries

A further review of the Intel386 ABI is needed.

6.1 C Library

6.1.1 Global Data Symbols

The symbols `_fp_hw`, `__flt_rounds` and `__huge_val` are not provided by the x86-64 ABI.

Chapter 7

Development Environment

No changes required.

Chapter 8

Execution Environment

Not done yet.

Chapter 9

Conventions

[1]

9.1 GOT pointer and IP relative addressing

A basic difference between the i386 ABI and the x86-64 ABI is the way the GOT table is found. The i386 ABI, like (most) other processor specific ABIs, uses a dedicated register (`%EBX`) to address the base of the GOT table. The function prologue of every function needs to set up this register to the correct value. The x86-64 processor family introduces a new IP-relative addressing mode which is used in this ABI instead of using a dedicated register.

On x86-64 the GOT table contains 64 bit entries.

9.2 Execution of 32bit programs

The x86-64 is able to execute 64 bit x86-64 and also 32 bit ia32 programs. Libraries conforming to the Intel386 ABI will live in the normal places like `/lib`, `/usr/lib` and `/usr/bin`. Libraries following the x86-64, will use `lib64` subdirectories for the libraries, e.g `/lib64` and `/usr/lib64`. Programs conforming to Intel386 ABI and to the x86-64 ABI will share directories like `/usr/bin`. In particular, there will be no `bin64` directory.

[1] This chapter is used to document some features special to the x86-64 ABI. The different sections might be moved to another place or removed completely.

9.3 C++

For the C++ ABI we will use the ia64 C++ ABI and instantiate it appropriately. The current draft of that ABI is available at:
`http://reality.sgi.com/dehnert_engr/cxx/abi.html`